OBM

A Solution to the Compatibility and Portability Problem

Göran Fries

CODEN: LUNFD6/(NFIP-3004)/1-24/(1978)

1978

LUND UNIVERSITY
Department of Computer Sciences

LUND UNIVERSITY
INSTITUTE OF COMPUTER SCIENCES
SÖLVEGATAN 14 A
S-223 62 LUND, Sweden

LUND, APRIL 1978

OBM

A SOLUTION TO THE COMPATIBILITY AND PORTABILITY PROBLEM

GÖRAN FRIES

CODEN: LUNFD6/(NFIP-3004)/1-24/(1978)

This work is supported by The Swedish National Board for Technical Development.(75-3354, 76-3609, 77-4426).

UNIV.
LEIDEN
BIBL.

C O N T E N T S

ABSTRACT 1

1. BACKGROUND ..... 1 - 6

1.1 INTRODUCTION 2
1.2 DEFINITIONS ..... 2 - 3
1.3 WHY PORTABILITY AND COMPATIBILITY ARE USEFUL ..... 3 - 5
1.3.1 PORTABILITY ..... 3 - 4
1.3.2 COMPATIBILITY ..... 4 - 5
1.4 ON VARIOUS SOLUTIONS TO THE COMPATIBILITY PROBLEMS ..... 5 - 6

2. THE O B M SYSTEM - A PROPOSED SOLUTION ..... 7 -16

2.1 GENERAL DESIGN AND USE OF SYSTEM ..... 7 - 8
2.2 COMPILERS ..... 8 - 9
2.3 THE OBMOS OPERATING SYSTEM AND COMMAND LANGUAGE ..... 9 -11
2.4 O B M ..... 11 -16
2.4.1 MACHINE CONCEPTS ..... 11 -13
2.4.2 INTRODUCTION ..... 13 -15
2.4.3 SEGMENTATION 15
2.4.4 PROGRAM SECURITY 16

3. IMPLEMENTATION ..... 17 -22

3.1 O B M AS AN ASSEMBLY LANGUAGE FOR DIFFERENT MACHINES 17 -18
3.2 ON THE EFFICIENCY PROBLEM ..... 18 -19
3.3 SOME INVESTIGATIONS ON IMPLEMENTATION ON UNIVAC 1108 ... 19
3.3.1 TRANSLATION PROBLEMS ..... 19 -20
3.3.2 PORTABILITY OF TRANSLATOR 20
3.3.3 EFFICIENCY OF OBJECT 21
3.3.4 CONCLUSIONS 22
3.4 IMPLEMENTING THE SYSTEM 22

ACKNOWLEDGEMENT 23

REFERENCES 24

ABSTRACT

This report gives a brief introduction to the compatibility and portability problem. Some methods wich have been proposed to solve these problems are discussed and one method is introduced. This method uses an intermediate language or a standardized machine. The method is somewhat related to the UNCOL idea [6]. The intermediate language or machine and implementation are briefly described.

1. BACKGROUND

1.1 INTRODUCTION

This report will discuss the compatibility and the portability problems. A definition of compatibility and portability will be given and some reasons why it is useful will be given. I will also present some of the methods used today to solve the portability problem and give a reason why these are not satisfactory. The main purpose of this paper is to propose a new method. This method will be briefly described. The method is built on a machine independent machine oriented language OBM. This language will be defined in a later report [10], a preliminary definition is already given in [9].

This report does not only discuss portability and compatibility of programs, but of jobs also including commands to the operating system. This is very important because it is much more difficult to change the commands in a command language than to modify the program written in a high level language. This is because the user may be a good FORTRAN programmer, but he might know nothing of operating systems, and to change from a "kind" system to a system like OS360 is not easy.

1.2 DEFINITIONS

Throughout this report the words job compatibility and job portability will be frequently used. I will now give my definition of these words by means of the following three criteria :

1) Ability to move high level language programs from one computer to another, possibly by means of macro processors.

2) Ability to run the same source deck(or equivalent) on different computers except accounting information.

3) In addition to the second criterion: The property that output depends only on input data and program and is independent of computer.

Digitized by the Internet Archive
in 2026

https://archive.org/details/IA41553421_0029

1.2 If the first criterion is satisfied we call it portability, while if the other two criteria are satisfied, we call it compatibility of different degrees from weak [2] to full [3].

1.3 WHY PORTABILITY AND COMPATIBILITY ARE USEFUL

1.3.1 PORTABILITY

The problem of portability is wellknown, and there is also a desire to solve it, particularly for software libraries. The problem should have been solved already, say by means of a standardized programming language. Tody there is such a language, "standard" FORTRAN. But if a non-trivial FORTRAN program is moved from one computer to another without any changes the result will be disastrous. There are a number of reasons why this will be the case. Some of the reasons are given below.

i) The two computers will not accept the same input format of the symbolic program. There may also be some minor restrictions on the syntax of some statements.

ii) The two compilers will not implement all statements exactly in the same way, because the language definition may be vague.

iii) There are undefined concepts in FORTRAN, such as: word length, methods of arithmetic, range of floating-point numbers, precision, character code, relation between integers (words) and string data, etc. There are no defined actions for underflow, overflow, divide fault, etc.

iv) In order to make compilers easy to write and to produce efficient code for a specific computer, dialects are created. This is sometimes done even if the first intention was to implement standard FORTRAN.

The first point (i) may easily be solved by simple pre-processing or better standardization.

The most common suggestion for solving the second (ii) problem is to use a subset of FORTRAN that is accepted and implemented in the same way by all compilers. This may be a solution if there is such a subset that is reasonable powerful. Does not such a subset exist ?

1.3.1 So far, it seems as if the CONTINUE statement is the only one which is not interpreted differently by different compilers! Of course there are slightly more powerful subsets that will be interpreted (almost) in the same way by most compilers.

The most serious of these points is the third one. Just imagine a "portable" FORTRAN program that runs correctly on a computer with say, 40-bit integers. On a computer with 32-bit integers the program may still run and give a result. Because of undetected overflow this result may be nonsense, but it may be presented to us without any warning! These problems are discussed in [1].

A standard programming language is not completely defined until it is implemented on some computer, as the implementation will finally define the language depending on and expressed in terms of the actual computer.

If all the concepts in (iii) above would be completely defined in the language, and without the possibility of programmer control, it may lead to inefficiency. It may force a computer to use a foreign arithmetic, and to act "against its own architecture".

A completely defined language may lead to still more problems of the type mentioned in (iv), if no methods for efficiency control are provided. A possibility of programmer control of "machine concepts" (wordlength,etc) built into the language, or given by parameters to the compiler, may be such a method.

1.3.2 COMPATIBILITY

Compatibility is often considered to be less useful and necessary than portability, but in the following situations, compatibility is clearly desireable.

1) Using one computer for test runs and for developing purposes, and another computer for production runs. Here it is important to have the same results, including errors, on the two computers.

2) Using different computers in a computer network.

3) Using another computer in case of overload or breakdown.

4) On exchanging software, including precision sensitive numerical algorithms, between users.

5) Wanting predictable results, independent of computer.

1.3.2 A compatible system is generally considered to be inefficient and expensive to use, but this is only partly correct. If it is possible to choose the degree of compatibility, from weak to full, we may control the degree of efficiency. The computers will also become faster and the loss in efficiency will be less expensive than the increase in manual work without access to compatibility.

If it is possible to choose compatibility when it is needed, it is useful even if it is expensive and inefficient to run. What is most desireable to be able to run a program at high costs or not to be able to run it at all?

1.4 ON VARIOUS SOLUTIONS TO THE COMPATIBILITY PROBLEMS

Much effort has been spent on solving the portability problems and some more or less good solutions have been presented. If the problem is to move a library or programming system from one computer to another then a language like FORTRAN mixed with "macro statements" may be used. The program may then be moved by help of a macro processor that expands the macros and produces a FORTRAN program for the actual computer. The macro must of course be defined for each machine. This method may be used to achieve portability, but since normal compilers are used the incompatibilities of these are still present. This may be useful to reduce (but only reduce) the manual work when implementing systems on "new" computers. ALTRAN, a portable language for formula manipulation, is an example of this technique [2]. The use of macro processors and pre-processors to achieve portability is very frequent [3, 4, 11].

A system aiming at a more extended form of portability has been developed in the Soviet Union, this is the ALMO system [5]. This has been used to make it possible to write programs without the programmer knowing which computer will eventually be used. ALMO is a language in which it is possible to express minimum requirements on precision and range for numbers. This will not give compatibility but rather good portability and the difference in results will in most cases be reasonably small.

The programming language PASCAL with its possibilities for type definitions is a somewhat "portable" language of this type. But far too many concepts are undefind to make it possible to write truly portable programs in PASCAL, and there is of course no compatibility.

1.4 A more advanced solution aiming at portability and perhaps a weak form of compatibility has recently been proposed by Dahlstrand [1]. This solution uses a new standard FORTRAN with existing compilers and therefor many of the problems still remains unsolved.

All these methods do only solve or try to solve the portability problem and no real attempt is made to solve the compatibility problem. There has been one attempt to solve it, UNCOL [6, 11] , but it is oldfashioned and has been considered a complete failure. What about the big manufacturers, are they not interested? It seems to me as if they are more interested in incompatibility than in compatibility. They have solved the problem for their own computer families, but only upwards. It is possible to buy a new and more expensive computer of the same make and still use the old programs with the "old results".

One method to achieve compatibility is to simulate one machine on another machine. This method is mainly used to develop software for a machine not yet constructed. It may also be used as a temporary solution when a computer center changes to a new machine. This method is rather inefficient and tailored for one particular application. If the simulation may be done in hardware or firmeware the method will be less inefficient and sometimes just as good as the old computer itself. The method proposed in this paper is somewhat related to this method of simulation. The difference is that the proposed solution will be a complete and flexible system that may be implemented by means of software, firmeware or hardware simulation.

Is it then impossible to solve the compatibility problem or will it lead to too inefficient programs? It is of course not impossible, but it is difficult. A solution will also make it possible (and easy) to produce very inefficient programs, very time consuming simulation may be necessary to produce the same result on every computer. The rather new field of microprogramming and dynamically micro-programmable computers may be very helpful in implementing compatible systems in an efficient way. The "OBM approach" to compatibility will be further discussed in the next chapter.

If the system will make it possible for the programmer to decide in what respects the result must be machine independent or not it will be possible to control the efficiency.

I am convinced that an advanced form of portability is absolutely necessary and that compatibility is important and with the further developement of computer networks it will be necessary.

2. THE O B M SYSTEM - A PROPOSED SOLUTION

2.1 GENERAL DESIGN AND USE OF SYSTEM

The OBM system is a system designed to make it possible to write programs and to construct a "job" completely independent of any particular computer. In order to use a computer to solve a problem, a sequence of OS commands, programs and data, on some media as punched cards, should be prepared and then submitted to any computer.

And we should have, independent of computer used:

ONE PROBLEM - ONE CARD DECK - ONE RESULT

This means full compatibility and is sometimes inefficient and expensive. In order to control the efficiency it must be possible for the programmer (user) to specify the degree of compatibility. All programs written according to the rules of the system are highly portable and any desired degree of compatibility may be achieved under full program control.

Most approaches to portability is to standardize the programming languages, but this has been at least a partial failure. Also in the OBM system standardized languages and data formats are important, but the system may accept several standards. Instead of standardizing the languages we will have a standard machine, and all programs are run on that machine. This machine is of course not an existing computer but a fictive computer with a very flexible architecture. This fictive computer may also be regarded as an intermediate language between high level languages and existing computers. This fictive computer or intermediate language is calld OBM.

The OBM system for compatibility is a system with the OBM language or computer as the heart, but also containing compilers from a number of high level languages to OBM, translators from OBM to a number of "host" computers and an operating system.

2.1

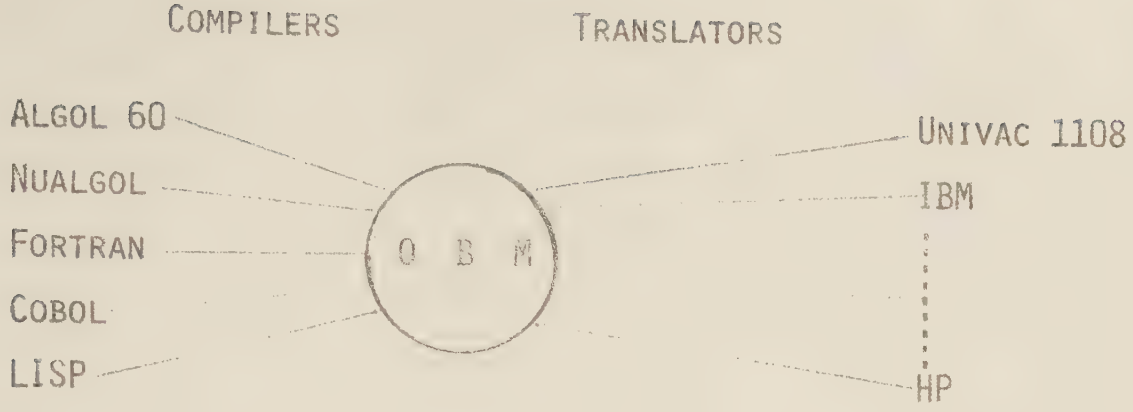

FIG. 1. THE 3-LEVEL CONCEPT OF OBM

The system may be divided into four parts:

1) Compilers from high level languages to OBM.

2) An operating system OBMOS, with a command language OBMOS-CL.

3) OBM - a machine independent machine oriented language.

4) Translators from OBM to existing machine languages.

2.2 COMPILERS

Let us look upon a compiler and the data to the compiler, the source program. The program is written in a standard programming language and it will be translated from this language into a machine independent intermediate language. All concepts of the high level language must be completely defined. That means that word length, precision and similar machine dependent concepts are prescribed. It must also be impossible to reach an undefined state because of an error. The effect of all statements in the high level language must be predefined, including errors. All errors must be detected. This may force the actual computer to act in an inefficient way.

Should the high level language be defined to make the result fit an IBM or UNIVAC or some other computer, or for the sake of justice, to fit no computer? The user should of course be able to choose. Parameters given to the compiler should control the produced code to fit some specified computer, but still the produced code should be possible to run on any computer. The specified computer could be "actual"and then the compatibility is lost, but not the portability.

Also the security and error detection may be inefficient, and the programmer should be able to switch off some checking by parameters to the compiler. Some errors like overflow and divide fault may be hardware detected and generates hardware interrupts. These interrupts take time only when they occur. Some computers does not have this feature, but this security is so important that the computers should have it. A computer without this feature should be forced to simulate it.

The compilers in this system are similar to those in other systems, but the compiler does not know the special properties of the computer.

The high level language is broken down to units expressed in the intermediate language OBM. This must be carefully done, in order not to lose the structure of the high level program. If this structure is lost a new level of inefficiency may be introduced, and the program cannot take advantage of the special features of the object computer.

The result of the compilation is a symbolic OBM program. The compiler must pass default values and parameter defined values for length, etc to this OBM program. The OBM program should also be "segmented" in a way to fit computers with quite different memory sizes and memory allocation techniques.

In the compatible OMB-system a great number of compilers will be implemented, and in order to facilitate this a multilingual technique may be used. A compiler consists of a language independent part, common to all compilers, and a language dependent part. Some work along these lines has already been performed by the research group in Novosibirsk [7].

2.3 THE OBMOS OPERATING SYSTEM AND COMMAND LANGUAGE

To the user, all machines are run under the operating system OBMOS. The interface between the user and OBMOS is the command language OBMOS-CL [8]. The operating systems of today seem to take care of the system efficiency and the system programmer, but they hand over many problems to the normal users. These users may be non professional programmers, knowing their problems but nothing of operating systems and secondary memory requirements.

2.3 Such programmers should be taken care of by the system, not the reverse. An operating system should be problem oriented, simple and with reasonably efficient defaults, but it should also provide the professional programmer with more complex tools.

One of the most important parts of OBMOS is the administration of secondary memories and their logical use. Secondary memories are organized in files (data sets), which are secondary memory areas with certain attached properties. Such properties are name, sequential or random, size, access rights, keys, format, etc. Files may be internal to the computer or external. External files must be given a standard format, to make them possible to move from one computer to another. In a compatible system external files are very important. Files may contain program or data. Program files may contain symbolic elements in different languages such as ALGOL, FORTRAN and OBM, but a program file may also contain "relocatable and absolute" elements. Symbolic elements are of course machine independent, but other program elements are machine dependent, but designed to fit the compatible environment.

In a compatible environment the transfer of secondary memories (such as tape) and sharing of data between different computers makes the standardization of data storage very important. Programs and data on external data media such as punched cards are also distributed among computers with different card readers and perhaps different code. Also line printers with different character representation and other hard copy peripherals may be used in the system. Much of these incompatibilities are impossible to solve by software and a standardization is needed, but much may also be done by software and the operating system should be able to handle this.

The fictive OBM computer should be capable of parallell processing and the use of many CPU:s. These features must be handled by the operating system, and if not present on the host computer, they must be simulated by the operating system. OBMOS does of course also include all other normal OS functions, like dispatcher, activity handler, user service routines, memory and processor handler, etc.

Such an operating system may be implemented on the host computers in various ways.

2.3

One impossible method of implementing an operating system in a compatible system of different computers with different existing OS, is to use the existing OS. The commands and needs expressed in OBMOS-CL can then be translated to commands in the OS of the host computer. Unfortunately it has turned out to be too difficult if we do not want to sacrifice compatibility. Some commands may be impossible to implement in an existing OS.

Another method is to implement the OBMOS system as the only operating system on all computers. This is a possible and good method, but the amount of work needed makes it impossible as a first solution.

Our first approach will be a combination of these methods. The OBMOS system is designed to be a real OS, but it is run under the existing OS of the host computer. OBMOS will setup a machine independent interface to the user and to executing programs, but it will use the OS-routines of the host computer to do the hard work.

The goal for OBMOS is to be the only OS, and in order to implement this the system will be written in a machine independent way as far as possible.

2.4 O B M

The OBM language is described in a very preliminary version [9]. A final, more complete and with fewer errors (?) will be published later as a report [10].

2.4.1 MACHINE CONCEPTS

In a compatible system such machine concepts as word length, and floating-point representation are very important to keep under control. The programs in a high level language are translated into the intermediate language OBM. In this language it must be possible to express such machine concepts as mentioned above. OBM is machine independent in the sense that it is possible to prescribe how the computer shall act and it is possible to run this on any computer. In some languages of today it is possible to describe the machine the program is to be run on, but in OBM it is possible to demand the computer to act in a prescribed way.

2.4.1 One of the main concepts of OBM is the type. We must use the type to describe the memory organization of the fictive OBM computer. The type describes the storage mode of data words. A type may express wordlength in bits, if this bitlength must be exact or if it is permitted to use longer words.

Not only data but also operators are dependent on type. For operators it is not only bitlength, but also word disposition, sign representation, etc. There are two logically different ways of using a type - one is to specify memory organization and on is to specify operators. All concepts in the first kind of type is also contained in the second kind, therefore only the second kind of type is introduced in OBM. This type is used for both purposes, but it must be noticed that even if a real type is used to define memory organization, a memory word does not have the property of being real or using truncating arithmetic but does only have a word length.

There are five main types: INTEGER, REAL, STRING, BIT and NODE. Each main type may be further described by word length, sign representation, character code, mantissa-base disposition, base, mode of truncation and storage mode. All main types cannot of course be given all these attributes, REAL is the most complicated one and BIT is the least complex with only a word lenght and storage mode. It is also possible to define a type that is machine dependent, thus it is possible for the programmer to specify the degree of compatibility.

As an example two real types are described (F and G):

```
F:TYP  REAL,8,24,16,B,C;
```

is a description of an IBM floating point number with 8 bits in the exponent, 24 bits in the mantissa, 16 as base, sign by special sign-bit and excess 64 for exponents, (given by B), and with truncating arithmetic (C).

```
G:TYP  REAL,ACTUAL;
```

will give the floating-point representation of the actual host computer, and is of course machine dependent.

A type is then used throughout the program to describe a data word or the action of an operation. The data word (computer cell) is not the only important feature of a computer, but also the memory structure and how this is built up from data words is important.

2.4.1 The OBM computer (language) may have a very complex memory structure - on the other hand if indexing should be possible it has to be well organized.

The OBM memory is divided into one or more data areas or pools, each containing only words with the same length (and storage mode). In order to define data areas a special POOL statement is used. An OBM program may contain several areas with different word lengths. Names of different words in a pool may be given by a DATA statement; also initial values other than zero (default) may be given in this statement. The words in a pool may be addressed by their names, with or without index. The index may be numerical (absolute) or a named data word (variable). If the words in a pool are defined to be of NODE type, also an indexing of partial words within the node is possible. All indexing may occur only within a pool.

The first part of each OBM program (segment) is the data and computer definition part containing the above described statements. Also other statements are included here to permit still more flexible machine descriptions.

2.4.2 INSTRUCTIONS

The other part of an OBM program is the executable part, containing the normal instructions.

OBM is not really a machine language, but rather a machin oriented language. The OBM statements are on a slightly higher level than ordinary assembly statements, but still the statements control only "primitive" functions of the computer. It is not possible to express complicated arithmetic expressions, the complexity of the arithmetic statements being on the level a:=b+c.

The instructions used in the instruction part can be divided into three classes:

i) Ordinary assembly instructions.
ii) Higher level instructions
iii) Operating system interface instructions.

The first class contains instructions for data transfer, arithmetic, shifting, logic and conditional branching. These are similar to ordinary assembly instructions, but somewhat more complex.

2.4.2 In most assembly languages there are only a few add instructions for a small number of types, but in OBM there is one add instruction for each possible word length and word disposition. In OBM add is not one instruction but rather a set of instructions where each member is choosen by specifying the operator (like add) with a type. This is the main feature that makes OBM to a machine independent language suitable for compatibility, but also a feature that makes OBM suitable for writing inefficient programs.

As the OBM memory may contain words of different length, and as it is desirable to performe arithmetic and data transfers between such words, this must be done in a well defined way. Words as operands with a length that differs from that of the operator must be expanded or truncated in a well defined way. This "trimming" of operands is defined in a way to make the result meaningfull in most cases. The trimming of real numbers will always give nonsens results, but always well defined nonsens.

The higher level instructions are instructions for loops, subroutines, string and stackhandling. There are a couple of reasons to introduce such instructions in OBM.

First, some machines may have hardware suitable for such instructions, and this hardware would have been impossible to use if only low level instructions were included in OBM.

Second, it is easier to represent a high level language by means of such instructions, without destroying the structure of the program. This means that it may be easier to produce efficient object code for some computers.

Third, it may also be easier to introduce security and security controls if such instructions are used. Security may here include controlled indexing, branching and parameter transfer.
The loops are not only introduced for efficient loop control, but the inner part of the loop may also be regarded as a form of block with some local variables and some protection. The protection of variables within loops are similar to that of PASCAL loops.

Subroutines are not introduced only as a "jump and remember return point" feature. The main purpose with subroutines is to define a sort of block unit that may be translated independent from all possible calling programs.

2.4.2 This is important when building program libraries and large program systems. It must be possible to translate routines and parts of a program separated from the rest of the program, and to translate routines to be used by many different programs. The data transfer to and from subroutines are well defined. OBM contains both recursive and nonrecursive routines. The recursion is made possible by a subroutine stack. To make an an secure expansion of OBM possible, OBM contains an external subroutine declaration. This makes it possible to write subprograms to an OBM program in machine code for the host computer. The OBM program takes no responsibility for what will happen outside the OBM program.

The stack and string handling are introduced to make it easy to use strings and stacks without to much care for character representation and pointers. Some computers have also special intructions for this.

It should be noticed that operating system interface instructions are included in OBM. A write instruction or an instruction to start up the execution of another program is just as much an OBM instruction as the add instruction. Some operating system routines may be regarded as complex machine instructions in OBM. This means that the operating system is in some respect an integrated part of the OBM computer (or language).

2.4.3 SEGMENTATION

As an OBM program may be written for any computer, the size of the OBM program must also fit any computer. We cannot of course restrict OBM to programs small enough for all computers.

One possibility is to implement OBM by a virtual or paging technique, but this may be inefficient. Therefore it is possible to divide a program into segments. The segment is the normal programming unit translated almost independent from other segments. Segments which have or have not been translated together are then gathered to build a program. A segment should not be divided into pages unless it is absolutly necessary. The OBM unit described in 2.4.1 - 2 is a segment.

2.4.4 PROGRAM SECURITY

Today the program security of most systems is very low. Security means that overflow, division by zero, data references outside data areas and similar errors must be detected. In most FORTRAN implementations an overflow may occur without any warning. On some computers certain errors will give a hardware interrupt, but other errors must be searched for under program control in a time consuming way.

All easily detectable errors, like arithmetic faults, should give hardware interrupts with a default action and a possibility for the programmer to take care of the interrupt himself. This is done in OBM. The default action is to give a warning and sometimes an error termination of the activity. But it is possible for the programmer to specify which interrupts he wants to take care of in a user defined way. This is done in a special "interrupt segment", and it is possible to make changes in this segment without any changes in the normal program segments.

Even if the user specifies a "no action" for a given interrupt, the checking will be performed during run time and a subroutine jump and return will be made. This may be very time consuming. For some errors it is possible to tell the OBM translator, by a pseudoinstruction, not to produce the code sequence for error detection. In this case it is useless to specify any interrupt routine for this error, because the error routine will never be entered. This production of error detection code may be turned on or off several times within the same segment.

3. IMPLEMENTATION

3.1 O B M AS AN ASSEMBLY LANGUAGE FOR DIFFERENT MACHINES

The idea of a compatible system implies that the system is implemented on several different machines. OBM may be regarded as a fictive machine that has to be simulated by the different machines in the system. OBM may also be regarded as an assembly like machine independent language. In this part of the report OBM will be regarded as an assembly language. The problem of implementation is then to write one translator for each machine in the system. Since OBM is on a slightly higher level than ordinary assembly languages, the assembler will be somewhat more like a compiler. As the language may specify concepts normally out of reach for the user the assembler will be much more complicated than a normal one. Today there is no computer with an instruction repertoir rich enough to implement each possible OBM instruction as one or a few machine instructions. In the future it may be possible to implement OBM as the machine language by means of the micro code (firm ware). At least it would be possible to implement machine instructions in a way as to make OBM to a suitable assembly language for the machine. Already today there exists some possibility to use dynamic microprogramming in the implementation of OBM. This will be further discussed in 3.2.

In order to construct a real flexible and useful system OBM assemblers must be written for a great number of different machines. An OBM translator is much more complicated (and larger) than an ordinary assembler. This is mainly due to the possibility to define the machine ("logical architechture") within the program.

The traditional technique in writing all these assemblers is far to difficult and time consuming to be used. It may be noticed that all these assemblers are translators from the same language, and only the machines are different. This means that the assemblers, or at least the language analysing part of them must be very similar to each other. It should then be possible to construct an assembler from one machine independent part and one machine dependent part.

3.1 The machine independent part can be used by all the assemblers in the system, and this part can be written in a machine independent language like OBM, directly or through a high level language. There is one important question, how much of the assembler is machine independent ? We do not have the answer today, but we may guess that the answer is at least 50 %. It may also be possible to write the machine dependent part such that the technique is somewhat machine independent.

In a compatible system and specially in a computer network also cross assemblers will be important. Also cross assemblers may be constructed using the above technique.

3.2 ON THE EFFICIENCY PROBLEM

The most common objection to the compatibility idea is that it may lead to inefficient programs. The efficiency problem and how to handle it on a high level has already been mentioned in 1.2.2. Even if it is possible for the high level programmer to choose the degree of compatibility and thus the efficiency, this does not completely solve the problem. It is still very important that the OBM translator will produce efficient object code. If we want full security we have to pay for it in form of time consuming checks. The only way of reducing this form of inefficiency is that the computer manufacturers take their responsibility and implement at least arithmetic fault detection by means of hardware (firm ware).

If the types are suitable to the actual computer, the generated code should be just as efficient as code produced in the normal machine dependent way. Also other types of arithmetic should be simulated in an efficient way. Some of these possibilities have been investigated, and for types not too strange to the computer the cost in efficiency is reasonable. For other types, particularly types with "long word length" or a "long exponent part" the cost in efficiency is great, but such types should be avoided and only used if absolutely necessary. Perhaps it is better to have an expensive possibility than no possibility at all for such types.

A way of solving some of the problems is to use the microprogramming facilities. If dynamical microprogramming is available, it is possible to implement special arithmetic by means of microcoded subroutines.

3.2

It is also possible to implement OBM as the "machine language" by means of micro-code. Microprogramming does not solve all the problems, but it gives a tool to speed up inefficient routines. The limited size of the writeable control store (WCS) in most computers is still a problem. Dynamical microprogramming makes it possible to give each OBM program a WCS contents of its own, and perhaps to change it infrequently.

3.3 SOME INVESTIGATIONS ON IMPLEMENTATION ON UNIVAC 1108

During the definition of OBM, a partial test implementation work has been performed in parallell. This work has been done not to construct a good implementation, but to construct a good definition of the language. Some impossible language elements and definitions have been ruled out, and some new concepts have been introduced. Also a work on a PASCAL-compiler written in OBM is going on. This work is a good test of the OBM language, and this has lead to some changes.

One of the most serious objections to the compatibility idea is that it is impossible to construct an efficient implementation. One reason for the test implementation is to prove or at least make it highly probable that an efficient implementation is possible, we do not want to hide the problems, and there are problems and some of them are very hard problems. Our test implementation will be further described in [12].

3.3.1 TRANSLATION PROBLEMS

The flexibility of OBM may cause the translator to be slow. If the translator searches for the most common form of a statement first the translator will be slow only for unusual constructions.
The security of OBM will also cause a time consuming check, but if security is wanted it must be paid for. From the test implementation we may derive that there is no reason to believe that the translator will be much slower than an ordinary assembler, if we notice that one OBM statement is equivalent to a small sequence of assembly statements and that the security is higher. If long wordlengths or concepts foreign to any computer are used, we have to accept a slower translation.

.3.1 Another and may be more serious problem is that the translator will be very large. This is caused by the many and often unusual possibilities in OBM. It may be noticed that the translator contains a part used by all OBM program translations, and a large number of routines seldom used. We do not yet know the size of the translator, but at least five times as big as a normal assembler should be a good estimation.

The solution of the problem is of course segmentation. This looks easy, but it must be noticed that the segmentation is dependent on what is a "normal" OBM program, and as this may vary from one instalation to another, and also from time to time, the problem is worse than we first thought.

Our test implementation shows that it is possible to construct a translator for OBM that is only slightly slower than a normal assembler and with a core resident part of reasonable size. But we have also seen that it will be possible to write OBM programs that are very slowly translated.

3.2 PORTABILITY OF TRANSLATOR

As the OBM translator for one computer will be large it is a hard work to construct one. This construction has to be made for all computers in the system. It should be very desirable to be able to write a portable translator, but as it is a translator to machine code the translator will be rather machine dependent.

To solve this problem the translator must be written in a machine independent way as far as possible, and then (a small) machine dependent part must be written for each computer and included in the translator. We have used this technique only to a small extent in our test implementation, but we have found that this technique is possible, but we do not know the size of the machine dependent part yet.

5.3.3 EFFICIENCY OF OBJECT CODE

Even if it is important that the translator is not too slow, it is fare more important to have a translator producing efficient object code. According to efficiency the worst statements are the arithmetical ones. The results from the latest implementation of arithmetic is as follows. If OBM uses the same wordlength and disposition as UNIVAC, the translator will produce the same code as if the statements were written directly in the assembly language. If the wordlengths are slightly changed the arithmetic will be about twice as slow, and if the word disposition is changed the arithmetic may be slightly slower. If the word lengths were greater than 72 (a double UNIVAC word) the problems began and execution was much slower. But if we compare the code produced by the OBM translator for such wordlengths to the code produced in the traditional way for programmed multiple precision arithmetic, there are no reasons why the OBM produced code should be slower.

The statements in OBM are more complex than in assembly languages. Arithmetical OBM statements performes a load from memory, the arithmetic and a store back to memory. A sequence of OBM statements may produce the following, store a register in memory followed by load the register with the same contents. This problem is not present in assembly languages but in compiled code from high level languages. This problem may be solved by a rather simple form of optimization, and as branching in OBM is very disciplined the optimization is rather simple. We have seen from the test implementation that an optimizing pass is necessary in order to produce efficient code. The removal of unnecessary load instructions is obvious and simple, but also other forms of optimization are interesting. This may sometimes be machine dependent, depending on available machine instructions some tests may be reversed and also the reversal of some loops may increase the efficiency significantly. This latter form of optimization is by no means simple or without danger. Also the use of registers and index registers in an optimal way may increase the speed.

3.3.4 CONCLUSIONS

If a program is written in OBM with types suitable for a certain computer, and then run on the same computer, we have not lost any efficiency of the object code. If more security is desired than available in hardware, we have to pay for this, but only for this. The program may then be run on another computer with fully compatible results, and we must pay for this by a loss of efficiency. In most cases this loss will be reasonably small.

It is also possible to write OBM programs with very peculiar arithmetic or long words. This is not directly available in assembly languages, it must then be programmed. Such a program will be very slow and inefficient compared to "normal" arithmetic, but this does not depend on OBM, it depends on the program and what is desired to be done. As OBM provides the programmer with such tools there may be a temptation to use them, and this may be expensive.

.4 IMPLEMENTING THE SYSTEM

The implementation of the intermediate language OBM has been discussed, but this is only one part of a complete system aiming at compatibility. When the system, containing compilers, translators intermediate language and an operating system, is defined it must be implemented. How should this be done ? The intension is to make the system complete including an operating system and then implement this as the operating system on a number of computers.

Does this mean that one sunny day all existing operating systems will be replaced by ours. It would be a great day ! But we are more realistic than that. We intend to implement our system to be run under the supervision of the existing operating system. To the user our system is the only operating system, but to the existing system it is just a program.

We intend to develop a machine independent, user oriented system good enough to be used when new operating systems will be constructed.

ACKNOWLEDGEMENTS

I wish to thank the members of our research group, Eric Astor, Ferenc Belik and Klas Sigbo, for their contributions to the discussions in the group.
I also want to thank Sten Henriksson for his reading of the manuscript and his valuable remarks from the "outside world".

REFERENCES

[1] I. DAHLSTRAND — Portabilitet inom teknisk vetenskaplig ADB.

[2] W. S. BROWN, A. D. HALL — Altran Users Manual & Altran Installation and Maintenace - Bell Laboratories, Murry Hill, New Jersey 07974.

[3] P. C. POOL, W. M. WAITE — Portability and adaptability. Lecture Notes in Economics and Mathematical Systems, 81. Springer 1973.

[4] P. J. BROWN — Macro Processors and Techniques for Portable Software. John Wiley & Sons, London.

[5] A description of the ALMO Language, published in russian: The Academy of Sciences of USSR.

[6] The Problem of Programming Communication with Changing Machines. CACM 1(1958) Aug., Sept.

[7] A. ERSHOW ET.AL. — Methods of Decomposition Synthesis and Optimization in Multilingual Programming (In russian). Preprint Novosibirsk, 1974.

[8] E. ASTOR — A Command Language for the OBM System. Report LU/CS 76-01. Department of Computer Sciences, Lund University, Lund, Sweden.

[9] G. FRIES — A Machine Independent Assembly Language. Report LU/CS 76-02. Department of Computer Sciences, Lund University, Lund, Sweden.

[10] G. FRIES — OBM - A Machine Independent Machine Oriented Language. Report to be published. Department of Computer Sciences, Lund University, Lund, Sweden.

[11] M.CAMPBELL-KELLY — An Introduction to Macros. MacDonald,London.

[12] G. FRIES — Implementation Problems of a Compatible System. Report to be published. Department of Computer Sciences, Lund University, Lund, Sweden.

- - - - -